Plan 9 Authentication in Linux *

Ashwin Ganti
University of Illinois at Chicago
`aganti@cs.uic.edu`

ABSTRACT

This paper talks about the implementation of the Plan 9 authentication mechanisms for Linux. As part of this effort I have implemented the capability device of Plan 9 as a character device driver for the Linux kernel, ported the authentication server to Linux and have written a PAM module (used by user space applications like `su` and `login`) that performs the authentication with the host owner's factotum.

1. Introduction

Traditionally, Unix authentication has been based on the passwd and shadow files kept in */etc* which contain both a hashed password and various information about the user (such as her home directory and default shell). Applications have to heavily depend on these files, hence it leads to less extensible code. Also if a new authentication mechanism is introduced it requires all the applications (like login, su etc) that use the authentication information to be rewritten to support it.

Pluggable Authentication Modules (PAM) [6] and Naming Services were created to provide more effective ways to authenticate users. PAM provides a generic framework enabling uniform authentication of user applications. It enables user applications to perform authentication without actually knowing the implementation details of the underlying authentication mechanisms. It is usually done through a password based authentication mechanism but also supports a challenge response interaction. An application can dynamically link a PAM module that implements a particular authentication scheme (ex: Kerberos). The advantage of PAM is that any change that occurs in the authentication mechanism does not require the applications to be retrofitted to support it.

PAM is a convenient way of authenticating the users in the perspective of an application but it operates in the same address space of the application and thus can be circumvented. Moreover, a malicious application can simply ignore the return values of PAM thereby defeating the effectiveness of the PAM framework. A PAM module does not have any increased level of privilege over the application that is using it. The onus of protecting the environment in which PAM operates is on the application.

In contrast, Plan 9's authentication is centered around a per user agent called *factotum* [4] following the lead of the SSH *agent* [8]. *Factotum* is a trusted process that holds the

*This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

secure keys of the user and negotiates authentication protocols on behalf of the user. Factotum is implemented as a file server in Plan 9 and applications that need authentication services communicate with factotum using the usual file system calls (read, write etc.) The applications need not be compiled with the authentication or cryptographic code and can remain agnostic of the underlying mechanism which is understood by factotum. Moreover the applications need not be retrofitted for the changes in the authentication protocols.

In Linux, applications like `su` and `login` have the setuid bit set to root so that they can operate with increased privileges (run as the super user [7]) needed to read a user's authentication information and to set the user id of the process. Once the user is authenticated, the application forks a new process and makes a call to the `setuid()` system call to change the user id of the process to the target user. Having an application run as setuid to root does not achieve necessary privilege separation [5] and if the application has been compromised then an attacker can get root access to the system. These applications however do not run as setuid to root in Plan 9. When a process such as the *login* program needs to change its identity it proves to the host owner's factotum that it has the required credentials by running an authentication protocol. Once a proof is established the process is given a capability to change its identity. This is a more secure way of handling things when compared to Linux.

My effort here is to achieve similar mechanisms for Linux so that security critical programs like `login` and `su` need not run as setuid to root. I have implemented the Plan 9 capability device for the Linux kernel, ported the authentication server as part of this effort.I have almost completed a working prototype of the port of `su` to use the cap device wherein I wrote a PAM module to perform the authentication with the host owner's factotum. The rest of the paper delineates the components needed in Linux to authenticate applications, explains the implementation details of the capability device, the `su` port and describes the steps needed to setup the Plan 9 authentication server for Linux.

2. Components needed in Linux

The components needed in Linux are briefly mentioned here and explained in detail in Section 4 including the implementation details.

2.1. Plan 9 from User Space a.k.a plan9port

Plan 9 from User Space [1] is a port of many Plan 9 programs from their native Plan 9 environment to Unix-like operating systems. Many of the Plan 9 utilities like the factotum and other user level file servers of Plan 9 are ported as user space utilities for UNIX like systems. This provides a Plan9 like environment in Linux so that programs like factotum and the authentication server can be used for authentication similar to the way it happens in Plan 9. It provides a useful platform for the applications like *su* to use the cap device.

2.2. Plan 9 authentication server for Linux

Each security domain in Plan 9 has a trusted authentication server where all the shared keys of the users are maintained. It also offers services for users and administrators to create and disable accounts, manage the keys etc. It comprises of two services *authsrv* and *keyfs*. *authsrv* is a network service which is similar to the KDC in Kerberos. It basically brokers the network authentication sessions. The *passwd* utility can be used by users to change their account passwords on the authentication server. *Keyfs* is a user level file system that manages an encrypted database of user accounts. I have

ported *authsrv*, *keyfs* for Linux using the *plan9port* services to achieve the native Plan 9 authentication in Linux. I also have ported *changeuser* which is a utility to create and manage user accounts in the authentication server. The side effect of this process was to create the user's accounts in the authentication server separately but this can be overcome by building a Name Service Module configurable through *nsswitch.conf* that retrieves the user's credentials from the authentication server.

2.3. Linux Capability Device

I have implemented the Plan 9 capability device for the Linux kernel as a character device driver. The capability device managed by the kernel is used to allow factotum to grant permission to a process to change the user id. It exposes two files `/dev/caphash` and `/dev/capuse`. In Plan 9, a process that wants to change its identity authenticates with the host owner's factotum. The host owner's factotum on successful authentication creates a capability and writes it to `/dev/caphash` and passes on the capability to the process which writes it to `/dev/capuse` to change the identity. Further details on the semantics and working of the cap device for the Linux kernel can be found in Section 4.1.

2.4. Pluggable Authentication Module (pam_devcap)

I have implemented a Pluggable Authentication Module *pam_devcap* that is an interface for the process to authenticate against the host owner's factotum. The user process passes the target user's credentials to *pam_devcap* which authenticates them against the host owner's factotum. *pam_devcap* uses the *p9cr* authentication protocol to talk to the host owner's factotum. It gets back from the factotum a capability in the form of a string which it stores for later use. Whenever the user id needs to be changed, the application requests the PAM module to do so which retrieves the stored capability and writes it to `/dev/capuse` after which the process runs on behalf of the new user.

3. Overview of the port

Figure 1 shows a sequence diagram of the overall flow between a user process that wants to change a user id, the PAM module and the host owner's factotum. The user process contacts *pam_devcap* initially to authenticate the target user's credentials. The PAM module contacts the host owner's factotum and talks to it using the *p9cr* protocol.

p9cr is a textual challenge-response protocol which is typically done between a factotum and a local program and not between two factotums as in the case of other authentication protocols in Plan 9. The protocol with factotum is textual where in the client writes a user name, server responds with a challenge, client writes a response, server responds with `ok` or `bad`. Typically this information that is being exchanged is wrapped in other protocols like *p9sk1* by the local programs before being sent over the network.

The factotum using the information from the authentication server validates the target user's information. If the credentials (user name and password) are valid then it creates a capability, hashes it and writes it to `/dev/caphash`. It also passes this back to the PAM module. Now the PAM module stores this capability for later use and sends back PAM_SUCCESS to the user process informing the successful authentication of the target user. When the process wants to change the user id it contacts the PAM module again which basically writes the earlier saved capability to `/dev/capuse` and returns PAM_SUCCESS. The application now runs on behalf of the target user.

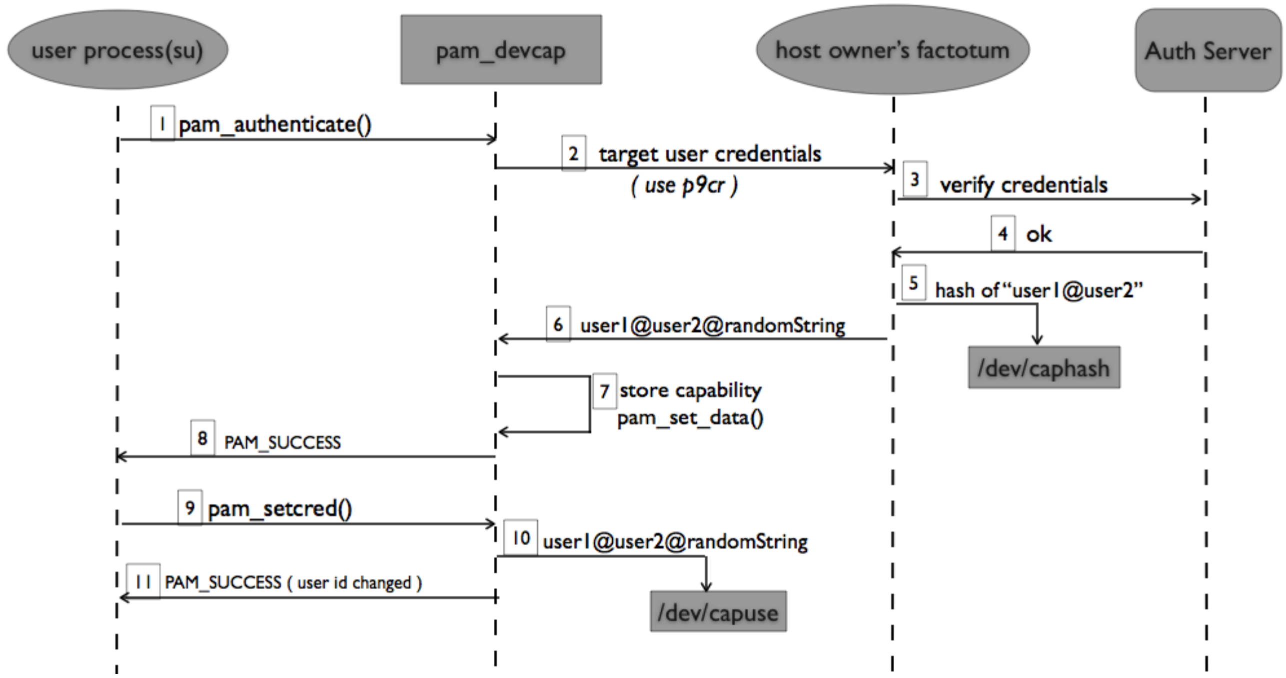


Figure 1: Overview of the port

4. Implementation

I shall now describe the implementation details of each of the components that are needed in Linux to use the Plan 9 authentication mechanisms.

4.1. Capability Device for Linux

Figure 2 shows the flow of control in the cap device.

1 A process running on behalf of *user1* sends a message to the the host owner's factotum requesting a capability to change its user id from *user1* to *user2*.

2 The host owner's factoum performs an authentication protocol(*p9cr*) with the requesting process to authenticate the target user's credentials provided by the requesting process.

3 Once the host owner's factotum knows that the requesting process has got the valid credentials of the *user2* it creates a capability of the form *user1@user2@randomString* and writes a HMAC SHA1 hash of it to `/dev/caphash`.

4 The kernel internally maintains a linked list of these hashes written by the host owner's factotum. It adds the newly created hash to the list.

5 The requesting process get the capability string and simply writes it to `/dev/capuse`.

6 Once the capability is written to `/dev/capuse` the kernel checks whether the process requesting the uid change is actually running on behalf of *user1*. If the process is not, then the kernel throws an error and the write operation fails.

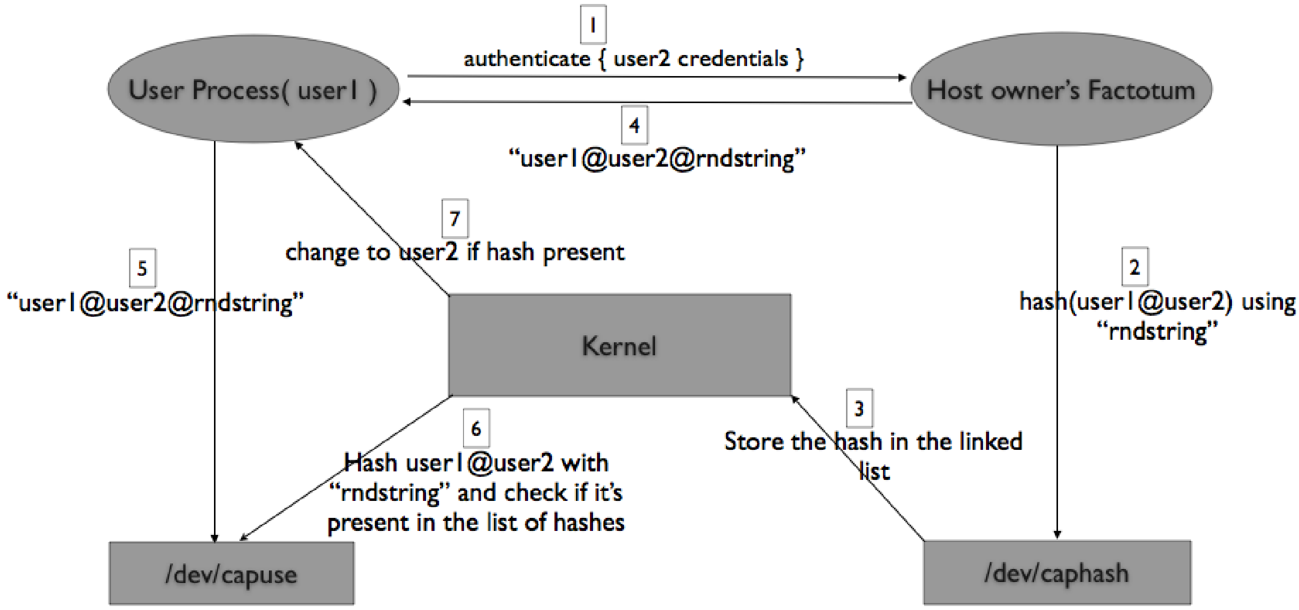


Figure 2: Flow of control in the cap device

7 The kernel now splits the capability string and creates a HMAC SHA1 hash of *user1@user2* with the *randomString*.

8 It does a linear search amongst the existing list of hashes for this hash and if it finds a match it changes the effective user id of the process to *user2*.

9 Once used the capability is discarded by the kernel i.e. it is removed from the list of the hashes.

10 If there is no match found or if there is a time out on the capability then the kernel just returns an error and the write operation to `/dev/capuse` fails.

4.2. Authentication server

I have made minimal changes to the existing authentication server code in Plan 9 and ported it to Linux using the libraries provided by *Plan 9 from User Space* in Linux. The authentication server as of now needs to be setup using xinetd or the like. This is only a temporary solution and I plan to avoid xinetd and setup the auth server in a more graceful manner in the future.

4.2.1. Steps to setup the authentication server under xinetd

1 Create the /etc/xinetd.d/authsrv file containing the following:

```
service authsrv
 {
        socket_type     = stream
```

```
                protocol        = tcp
                wait            = no
                user            = root
                server          = /usr/local/plan9/bin/authsrv
                server_args     = -d
        }
```

2 Create an entry in /etc/services

```
authsrv     567/tcp         # Plan 9 authentication server
```

3 Make an entry in $PLAN9/ndb/local for the auth server

```
authdomain = <domain of the machine running the auth server>
  auth = <host name of the machine running auth server> port = 567
```

4 Make sure the `$NAMESPACE` environment variable is set. This is very essential for the authentication server to run. Typically it defaults to `/tmp/ns.$USER.$DISPLAY`. Set it manually if not set by default using the following command :

```
export NAMESPACE=<directory path>
```

5 Restart xinetd

```
sudo /etc/init.d/xinetd restart
```

4.2.2. User Accounts' setup on the auth server in Linux

The authentication information of the users in Linux is generally handled by storing them in /etc/ files or some kind of naming service (configurable through *nsswitch.conf*). But the authentication protocol(*p9cr*) that happens between the host owner's factotum and the process is brokered by the authentication server. The authentication server in Plan 9 stores the shared keys of the users including the account information. Since the Plan 9 shared key protocols(*p9cr*) are being used to authenticate the users to the host owner's factotum it is necessary for the existing user's accounts to be created again in the authentication server. I have ported the Plan 9 *changeuser* utility to help add user accounts to the auth server.

```
changeuser <userid>
```

Users are represented by integer uids in the Linux kernel unlike the Plan 9 case where the users are represented as string indentifiers. The capability that is written to `/dev/caphash` and `/dev/capuse` is of the form *user1@user2@randomstring* where *user1* and *user2* are the string values of the user identifiers in the case of Plan 9. User identifiers are

strings in the Plan 9 kernel. NFS solves a similar problem by having a user space mapping daemon that is contacted by the kernel to get the integer user ids for the string names that are supplied. The easiest way to solve this problem in our case was to simply create the user names in the authentication server by the string equivalents of the uids (for example: *changeuser 1000*). Now when the capability is created by host owner's factotum it would be the actual integer userids in string form that the kernel can understand. Whenever the capability is written to the kernel it is converted to an integer. Since the kernel understands the integer equivalent of this it simply changes the userid of the process to the target user id thereby avoiding the requirement of a user space mapping daemon. Again this is a temporary solution. It would be nice to have a user space mapping daemon in the long run.

4.3. Pluggable Authentication Module

The PAM module *pam_devcap* authenticates against the host owner's factotum, retrieves the capability and whenever the user space application wants to change the user credentials, writes the capability to `/dev/capuse`.

4.3.1. *pam_devcap* configuration with PAM framework

1. Copy the module's shared object file `pam_devcap.so` to `/lib/security`.

2. Include the following line in the beginning of the PAM configuration file for the application using this module - `su` in our example.

   ```
   auth     requisite    pam_devcap.so
   ```

   `auth` signifies the PAM module's interface type. Modules with this interface type authenticate the user by typically asking a password. The module also can change the user's credentials such as Kerberos tickets or group memberships.

   `requisite` is the control flag for the *pam_devcap* module. It tells PAM what to do with the result of the module(pass/fail). Since PAM modules can be stacked in a particular order, control flags decide in what way does the result of the current module affect the final authentication outcome of the user. The `requisite` flag tells PAM that the module has to return success for the authentication to continue. The user is notified immediately if the module fails.

   `pam_devcap.so` is the name of the module that implements the PAM interface.

3. `pam_devcap.so` implements the service module's equivalent methods for the `auth` interface type i.e. *pam_sm_authenticate()* and *pam_sm_setcred()*. In *pam_sm_authenticate()* the PAM module authenticates against the host owner's factotum and retrieves the capability for the user. The capability is used to change the user id in *pam_sm_setcred()*.

4.3.2. Logical Flow of authentication using PAM

Refer to Figure 1 for the overall flow.

1 The application makes a call to the *pam_authenticate()* supplying the target user's credentials to the PAM module.

2 The PAM framework internally routes it the service module's (i.e. *pam_devcap*) implementation of the pam_authenticate() method which is *pam_sm_authenticate()*

3 The authentication happens in the pam_sm_authenticate() method of the PAM module.

4 The target user's credentials are passed over to the host owner's factotum which validates them by contacting the authentication server. Upon successful authentication it returns the capability as a string to the PAM framework.

5 The PAM module saves this capability using *pam_set_data()* and returns PAM_SUCCESS to the user application.

6 When the application wants to change the user id it makes a call to the *pam_setcred()* function.

7 The service module equivalent of the *pam_setcred()* is the *pam_sm_setcred()* function which is internally called by the PAM framework. The PAM module retrieves the capability that it saved earlier using the *pam_get_data()* function. It writes this capability to `/dev/capuse` which results in the kernel changing the user id of the process to the target user id of the process.

5. Future Work

Although `su` has almost been ported to use the capability device there are some features that need to be implemented in the long run to make the Plan 9 mechanisms to be used in a more graceful and complete manner in Linux. The authentication server that is currently ported does not have the *ndb* database file that implements the speaks-for relationship for the host id. Also the authentication server is currently setup to run with xinetd or the like. It would be useful to make it run as an independent service. It would also be nice to have a user space mapping daemon that maps the string user names and the integer user ids. This daemon would be contacted by the kernel to resolve the user names it gets after parsing the capabilities written to the device files (by the process and the host owner's factotum). Lastly, since Linux uses naming services to get the login information of the users it would be useful to have an NSS module that retrieves the user information from the authentication server and have it configurable through *nsswitch.conf* so that applications can directly make *getpwnam()* like calls to retrieve the user information from the auth server.

6. Related Work

Privelege separation by running a separate process for each user is widely used. `qmail` [2] forks off a new process(and then does a setuid) for each user to deliver email.

Linux implements the concept of Compartmented Mode Workstations that have split up root privileges into about 30 separate capabilities [3], including a *setuid* capability. The programs do not run as root and assume priveleges when needed and drop them when they don't thus implementing least privelege.

SELinux provides protection by limiting the priveleges of a process based on the user on behalf of which the process runs and also the process's executable.

In comparison to these methods the *cap device* provides even fine grain one time use capability to limit the setuid priveleges further by allowing only the host owner's factotum to create the capability and managing the capability system in the kernel thereby providing better security.

7. Conclusion

I have almost completed the port of `su` to use the Linux capability device. I had to make very minor code changes to remove the *setuid* system calls from `su`'s code. Since the PAM module abstracts out the internal details of the authentication mechanism the port is practically a no-op. Fortunately one of the various *su* implementations available uses PAM for authentication. `su` can now run as an unprivileged user instead of being setuid to root.

This proves that the capability device for the Linux kernel is a very useful authentication mechanism considering the existing alternatives. Programs like `su`, `login`, web servers , mail servers etc. need not run with the setuid bit set to root.

Since these applications run as root an attacker can get root access to the system by exploiting any kind of security flaw in the application (buffer overflows etc.). By implementing this new authentication mechanism even if these applications are compromised we do not give away much when compared to the former case. This achieves better privilege separation thereby providing better security.

References

[1] Plan 9 from user space. `http://swtch.com/plan9port`.

[2] Qmail. `http://cr.yp.to/qmail.html`.

[3] Jeffrey L. Berger, Jeffrey Picciotto, John P. L. Woodward, and Paul T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Trans. Softw. Eng.*, 16(6):608–618, 1990.

[4] Russ Cox, Eric Grosse, Rob Pike, David L. Presotto, and Sean Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, Berkeley, CA, USA, 2002. USENIX Association.

[5] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. *12th USENIX Security Symposium*, August 2002.

[6] Vipin Samar. Unified login with pluggable authentication modules(pam). In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10, New York, NY, USA, 1996. ACM Press.

[7] W. Richard Stevens. *Advanced programming in the UNIX environment*. Addison Wesley Longman Publishing Co., Inc., 1992.

[8] T. Ylonen. SSH - secure login connections over the internet. Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA):37, 1996.